Software Quality: From Legacy Codes to Web-based Enterprise Applications

ARMIN PRUESSNER
CHRISTOPHER PATERNOSTRO
National Oceanic and Atmospheric Administration, National Ocean Service

The abundance of legacy software still in use today is astounding. Many software modules have their origins 20 years or earlier and have been reused, reengineered, or integrated into other applications with little expectation by the original authors that the module would be used beyond the original scope. The reasons for maintaining legacy software beyond its expected lifetime are numerous, including application expertise by the original author and prohibitive costs of rewriting. Furthermore, the original source code has already been certified and quality controlled by internal experts, so the time required to rewrite the code may become prohibitive.

Today, many legacy applications are moving to the Web environment to take advantage of an easily maintainable and accessible interface and to allow for universal access. Furthermore, the applications use an enterprise framework involving centralized servers, databases, client Web front ends, and various other subsystems. This article discusses some of the issues in developing quality software systems when integrating legacy software into Web-based enterprise applications. The main issues are discussed and examples are given using a case study in the oceanic data processing field.

Key words: enterprise applications, legacy codes, oceanic data processing, reengineering, scientific computing, Web applications

INTRODUCTION

Quality pioneer W. Edwards Deming once said: "It is not necessary to change. Survival is not mandatory." By this, of course, he meant that change is inevitable. Unfortunately, change is not always immediate, even in the rapidly changing area of software development. This is particularly evident in the abundance of legacy software still in use today. Legacy software is sometimes referred to as "any information system that significantly resists modification or evolution" (Brodie and Stonebraker 1995) or in practical terms as "software that is vital to our organization, but we don't know what to do with it" (Bennet 1995). Legacy systems often run on antiquated hardware and are difficult to maintain since the original developers are usually not with the organization anymore, which may lead to an overall lack of understanding of the internal workings of the program. Nonetheless, there are numerous reasons legacy code is still in use today in many organizations. The main reason appears to be that they are "fully functional systems that are *required* to support the ongoing businesses of the institution" (Laverty et al. 2004). Within the authors' organization, managers often reasoned that:

- The software functions satisfactorily so there is no need to replace it.
- The software still provides a recognized scientific and technical approach for mathematical analysis.
- The user environment (hardware and interface) has not changed.

- The cost of developing a new system from scratch is prohibitive.
- The system is not well understood and is either not fully documented or documentation has been lost.
- The system was originally designed by subject-matter experts whose expertise was lost with their departure.

Instead of rewriting legacy code from scratch to allow for universal access, legacy software is often integrated into Web-based enterprise applications, which are hosted on a server and simultaneously provide services to a large number of users over a computer network. The integration of legacy software provides challenges in terms of software quality. The authors discuss some of the key issues in designing a quality enterprise software application involving legacy codes. They illustrate some of the key concepts by giving examples from their own scientific computing application. The software being developed is an oceanic data processing system for the National Ocean Service (NOS) at the National Oceanic and Atmospheric Administration (NOAA), which involves numerous subsystems, including a Web interface, databases, and legacy components. The legacy components were developed by subject-matter experts and are difficult to redevelop for many of the reasons mentioned previously, as well as the focus of the expertise changing from pure mathematics to ocean science.

APPLICATION OVERVIEW

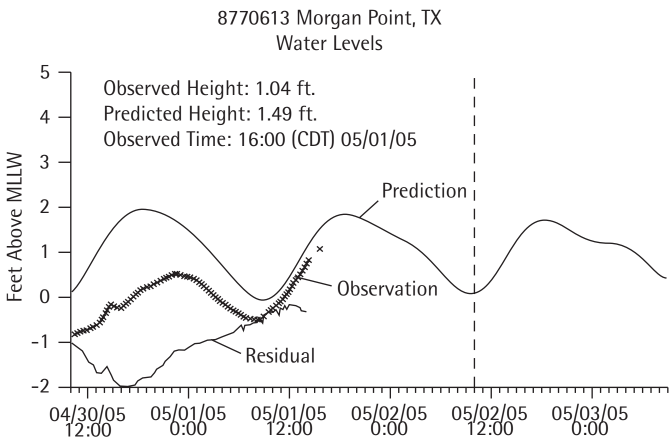
Oceanic Data Processing

The Center for Operational Oceanographic Products and Services (CO-OPS) at NOS has the mandate to provide the nation with accurate observed water-level information and tide and tidal current predictions for the U.S. coasts. Sensors on the ocean floor or side-mounted sensors on bridges, piers, or other structures provide data on water levels and current velocities (speed and direction), as well as meteorological information. The incoming data are quality controlled and analyzed to provide curve fit coefficients to make predictions of future water currents and levels. Traditional tidal harmonic analyses of the observations are used to compute the coefficients in response to forcing of the Earth-moon-sun gravitational system. These coefficients are then used in a tidal prediction equation to create tidal predictions (Schureman 1940). Figure 1 shows a resulting plot of water-level predictions obtained via the analysis routines.

Currently, data processing is done either manually by oceanographers on a workstation interacting with database queries, file transfers, running analysis routines, and other individual applications or semiautomatically for certain Web-based product applications. Depending on the application, the processes involve sequential use of separate programs and procedures and, often, separate machines. Some of the more intensive manual procedures require oceanographer expertise to analyze and accept results (for instance, the procedure for analysis of the seasonal constituents and the process for rejection of certain low amplitude constants for use in the tide prediction equation).

Although the software is reliable, the *process* is discontinuous, disjointed, and error prone because of complex input requirements (that is, complicated user-generated control files) that are inefficient for handling large volumes of data and make troubleshooting and diagnostics very difficult. It also results in potentially inconsistent results depending upon the interface. Separate applications of the legacy code have been developed by separate oceanographers and information technology (IT) groups with minimum collaboration until recently. With the anticipation that a large number of sensors will be deployed in the near future, a decision was made to integrate the software modules into a universally

FIGURE 1 Water level predictions using legacy analysis routines



accessible Web-based enterprise application known as C-MIST (see Paternostro, Pruessner, and Semkiw 2005). The authors' solution approach involves all analysis routines as used previously, but makes use of a *wrapping* technique, which "wraps an existing component in a new more accessible software component" (Bisbal et al. 1999). The technique of wrapping as opposed to redevelopment has significantly less impact on the underlying system (Bisbal et al. 1999) and thus is attractive for the project.

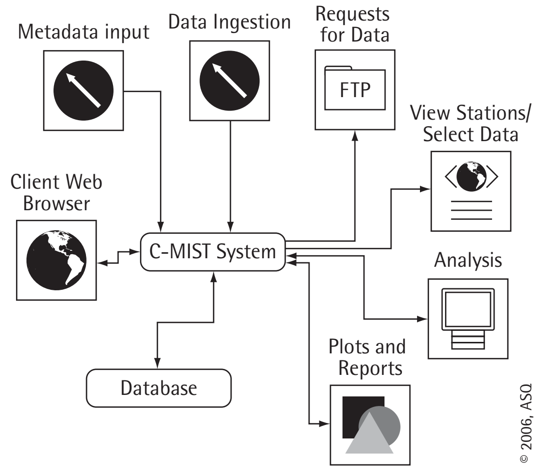
Subsystems

The system consists of various modules. A rough schematic of the software architecture is shown in Figure 2. The software suite includes a Web-based user interface, a database to store the raw data and analysis results, analysis routines to generate the predictions, plotting and reporting routines, interfaces to select data, modules to input metadata, as well as tools to request data (via FTP or Web services). The analysis tools correspond mostly to legacy FORTRAN software (Shureman 1940; Zervas 1999), which are treated as black box modules.

Legacy Components

The curve fit analysis modules to generate the predictions involve complex mathematical routines (harmonic decomposition analysis) developed many years ago by experts in the field. At that time, they went through rigorous quality control measures and have been approved by NOS for use in analyzing data and disseminating results to the public. Unfortunately, they are difficult to redevelop due to lack of subject-matter expertise, plus source code is poorly documented and therefore difficult to follow. There is also limited capability to reengineer the source code directly. Only limited documentation for the previous updates of legacy code and procedures for harmonic analysis and prediction exist (Zetler 1982). It should also be noted, that redevelopment at this time is not practical since the original software has already gone through the rigorous approval process and quality control would take a prohibitively long time, given the urgency of processing large volumes of incoming data.

FIGURE 2 Software architecture of the C-MIST sytem



SOFTWARE QUALITY MODEL (CMMI)

The key to generating quality software (including software involving reengineering or integration of legacy code) is to apply proven quality principles both for products and processes. This is facilitated by choosing a quality model and having all project team members follow the structured model guidelines closely. The key notion of importance is not so much the particular model—there are many equally effective models—but rather that a proven quality model is chosen. This helps guide the project team in being *proactive* in making quality an integral part of the product and process early on, rather than reactive when problems occur. Clearly the cost of quality rises exponentially the later the error is found (Schulmeyer and McManus 1999; Pressman 1997). Thus, it is important to include quality principles early in the project life cycle.

The authors chose the Capability Maturity Model Integration™ (CMMI) (SEI 2005a) in its *staged representation*. In particular, they focused on the CMMI for software engineering. The staged representation involves five maturity levels ranging from ad-hoc reactive procedures to proactive procedures focusing on continuous process improvement. Figure 3 comes from SEI 2005b and shows the various maturity levels illustrating the differences between an unpredictable, reactive environment (Level 1) to a controlled environment focusing on continuous improvement (Level 5). Note CMMI is applicable to all areas, not just software engineering. Also, it should be clear that CMMI involves technical procedures as well as management techniques to achieve a quality product.

The project team at NOAA/NOS/CO-OPS has attained CMMI Level 3 and is applying these procedures to the C-MIST project. In particular, many CO-OPS projects previously did not follow any framework at all: basic requirements were documented and then developers started writing code based on these (minimal) requirements. The work would be presented to the customer, who would come back with more (or new) requirements. The process would iterate until a stable product emerged that the customer was happy with. This ad-hoc setup is clearly not efficient for a large-scale project such as C-MIST.

By following the CMMI, the authors focused on a structured methodology combining planning, management, development, testing, documentation, and training to help deliver a quality product. They started by generating project management plans, which allowed for tracking of costs and schedule. These plans allowed for repeatable successes on previous projects and were deemed applicable to C-MIST. Furthermore, the authors defined standard techniques and planning documents for various phases of the software life cycle, including requirements elicitation, software requirements specifications, evaluation techniques for third-party software, physical designs, testing plans, and long-term maintenance.

This article will not describe the main techniques of CMMI, but will focus on integration of legacy code within a Web-based enterprise framework. It should be clear, however, that techniques as described in (CMU 1994; SEI 2005a; SEI 2005b) give a structured good framework, which can be beneficial for large-scale projects. The authors are making use of many of the techniques in striving to meet their own project goals.

LEGACY CODE SOFTWARE QUALITY

During requirements elicitation and later during conceptual design it became clear that the key to software quality involving the legacy code module involved the following main steps:

1. Verify the results of the legacy code running in a new environment (hardware/software).
2. Write a generic application program interface (API) wrapper around the legacy code to facilitate communication to and from the enterprise application framework.
3. Design an enterprise application framework using modular subsystems.
4. Design a generic API from the enterprise application to the subsystems so that the subsystems can be replaced in the future.

For the authors' application, verification in step 1 was done using data sets and results of previous analyses. The reproducibility of these results and the evaluation by the users of the routines were used for verification.

For some applications source code is not available, so step 1 would not be an issue. In this case, the application still runs under the original hardware/software environment. However, all other steps listed previously still apply. In the authors' application they were able to harvest the original source code and port it from the original environment (64 bit SGI) to the new one (32 bit Linux).

FIGURE 3 Staged representation of Capability Maturity Model Integration™

Staged

Maturity Level 2	Maturity Level 3	Maturity Level 4	Maturity Level 5
7 Process Areas	14 Process Areas	2 Process Areas	2 Process Areas
• Requirements Management • Project Planning • Project Monitoring and Control • Supplier Agreement Management • Measurement and Analysis • Process and Product Quality Assurance • Configuration Management	• Requirements Development • Technical Solution • Product Integration • Verification • Validation • Organizational Process Focus • Organizational Process Definition • Organizational Training • Integrated Project Management • Risk Management • Integrated Teaming • Integrated Supplier Management • Decision Analysis and Resolution • Organizational Environment for Integration	• Organizational Process Performance • Quantitative Project Management	• Organizational Innovation and Deployment • Causal Analysis and Resolution

Runtime Behavior of Legacy Code in New Environments

When running legacy code, which often has been tailored to a particular hardware/software environment, care must be taken when porting to a new environment. Beyond basic issues of compiling the source on the new environment, one must verify that the software produces the same results. In particular, the authors want to confirm two critical verification criteria: eliminating environment-dependent logic and robust numerical precision.

Eliminating Environment-Dependent Logic

The first issue involves logic due to if-then blocks, where the criteria are based on floating point values. The authors want to make sure the software behaves the same and business logic is not dependent on precision. Usually, in order to accurately verify logic, a design specification is necessary. In this case, however, a design specification likely never existed and one is left to examine the code manually. Environment-dependent logic is often an issue in poorly written code involving scenarios such as that found in Table 1. In the given example the idea is to proceed with one part of business logic if the variable is essentially nonzero. It should be clear from the code listed that depending on numerical precision of the floating point variable *f* and computations involving *f*, logic could change depending on hardware/software, that is, the software may or may not end up within this block of code. This logic becomes hardware/operating system dependent, for example, 64 bit computing environments have a potentially higher precision rate than 32 bit environments, so *f* may attain positive values less than the criteria 1.0E-20 given. In that case, even though the intent was for nonzero positive values to enter that block, for this given environment one may not proceed with the block of business logic.

TABLE 1 Example of environment dependent logic

```
Float f;
If [ f > 1.0E-20 ] then
     Do something…
End if
```

© 2006, ASQ

The authors' experiments with porting C-MIST legacy source to a new environment (from 64 bit SGI to 32 bit Linux) involved running numerous and varied input data sets in batch mode and comparing the resulting output parameters. They generated plots of water speed and direction predictions to determine if they produced similar results. The intent at this step was to verify logic, not to determine how closely the result matched the expected result, that is, did they traverse the same code and receive the expected result type? This can be thought of as a pass-fail type of test. All of the ported code performed as expected.

Robust Numerical Precision

Another issue is numerical accuracy, that is, are the results the same (given a relative tolerance) on the different platforms and operating systems? In order to verify numerical accuracy and consistency, one can measure the relative error of results obtained in the old environment to the new one. Let r_{err} be the relative error and s_{old} and s_{new} be the solutions obtained in the old and new environments, respectively. One can define the relative error as:

$$r_{err} = \frac{s_{old} - s_{new}}{1 + |s_{old}|} \qquad \text{(Equation 1)}$$

This representation of the relative error may differ from some textbook definitions (see Burden and Faires 2005). The representation in Equation 1 is also useful since it allows for a continuous representation of the relative error and balances the error between a relative (for large values of s_{old}) and an absolute (for small values of s_{old}) scale. Adding one to the denominator is necessary since s_{old} may attain the 0 value in which case the relative error is undefined. In some cases the one in the denominator may also be replaced by some positive value α greater than machine precision. Although the absolute error is often appropriate for data sets where all values have roughly the same order of magnitude, for values covering a wide range, the relative error often provides more useful information. See Mittelmann and Pruessner (2006) for a discussion on metrics in error analysis and choosing appropriate error measures in comparisons.

If the authors' measured data have only two significant digits (as is the case here), they cannot expect to have a higher precision of the solution than two significant digits. Significant digits are the number of digits of accuracy of a number when specified in exponential notation with leading nonzero (for example, 1.234 E-10 has four significant digits and 0.923 = 9.23 E-1 has three significant digits). Being aware of significant digits is important for the following reason: Although computers can store numeric figures down many decimal places it does not mean that all those decimal places are significant. During the authors' numerical verification of legacy software, they looked at the predicted speed and direction results for various oceanic data sets and compared the results of the software running in the old environment to the new. The relative error was computed as in Equation 1. Figure 4 shows the relative error of computing curve fits for water speed. The plot shows that less than one percent of values had any error at all (99.78 percent had no error), with the largest measurable error about 1.57 E-3 and an average error of 5.31 E-7. For plots of this type, where a larger percentage of samples have nontrivial (nonzero) errors, ordering the relative errors by size may give more insight into the data. In the authors' case, however, it is clear that few samples have any error at all so that no additional useful information can be extracted using this method. In any case, the authors illustrate that their software performs as expected. For a more detailed description of numerical precision and round-off errors see Burden and Faires (2005).

FIGURE 4 Relative errors moving to new hardware/software environment

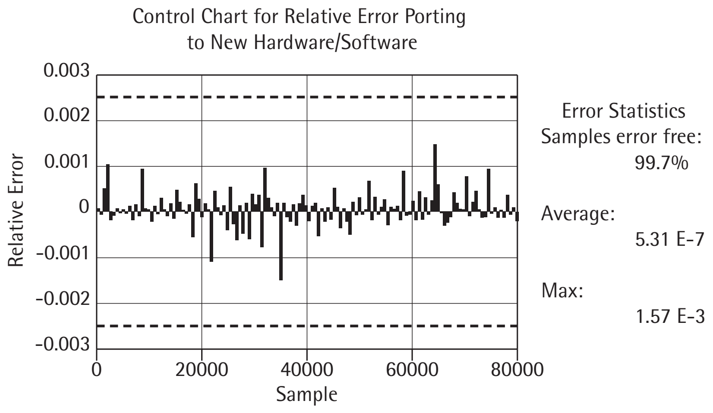
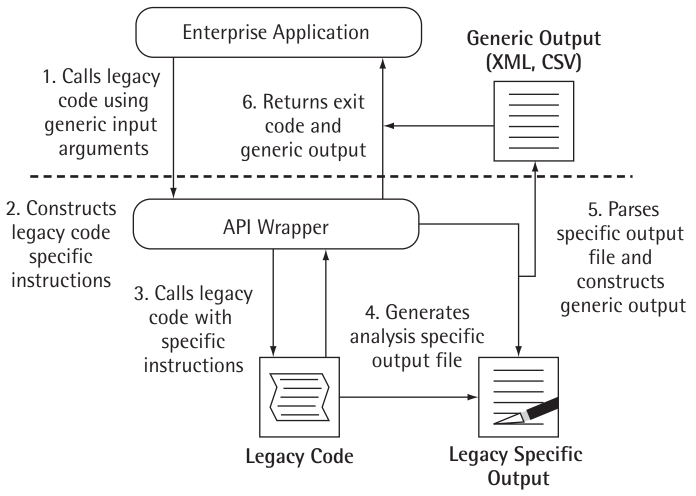


FIGURE 5 API to legacy components


Enterprise Application
Generic Output (XML, CSV)
1. Calls legacy code using generic input arguments
6. Returns exit code and generic output
2. Constructs legacy code specific instructions
API Wrapper
5. Parses specific output file and constructs generic output
3. Calls legacy code with specific instructions
4. Generates analysis specific output file
Legacy Code
Legacy Specific Output

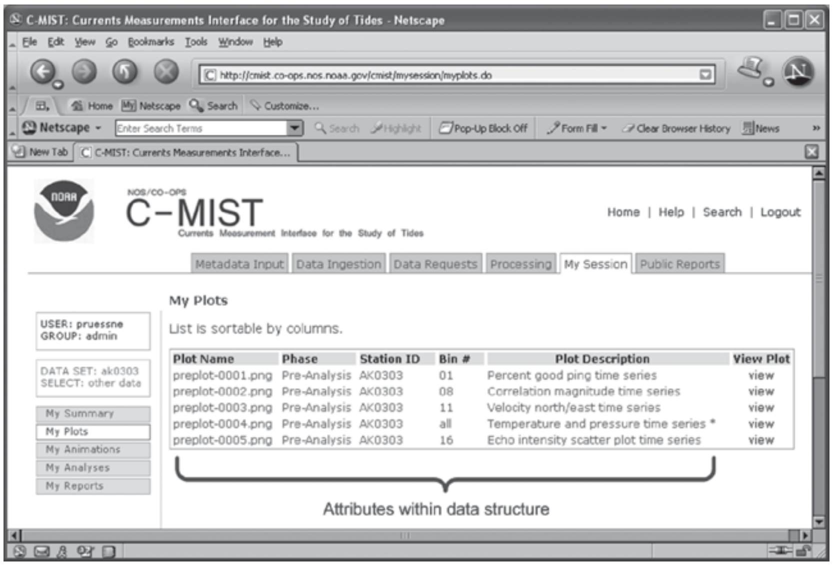
Generic APIs to Legacy Code

In order to encapsulate legacy components and possible quality defects, a generic API or programming interface to the legacy code must be written. The API wrapper accomplishes several things. First, it enables communication to the legacy module in a uniform manner. Second, it handles all error and exception handling consistently so that potential problems within the legacy module can be isolated and captured directly via the API wrapper. Finally, generic APIs with exception handling are good programming practice for modern applications and enable reengineering and reusability for other applications.

The C-MIST application communicates with the legacy code via a generic API wrapper. This involves a set number of arguments such as the data input files, data output files, and a control file name containing legacy subsystem specific options. This wrapper calls the legacy subsystem and verifies that the module ran successfully. It also verifies that the appropriate output file was generated. Furthermore, the wrapper parses the output file to generate a generic-format output. The output file may be in comma-delimited (CSV) or extensible markup language (XML) format and contains the attribute name and value. Generic output formats are a good idea since they enable enterprise applications to make use of results regardless of the underlying subsystem and output format produced.

Figure 5 illustrates the API to the legacy application with step-by-step instructions. The enterprise application communicates with the legacy code via an API wrapper with generic input arguments (step 1). The API constructs any legacy code-specific instructions (such as a control file) and calls the application (steps 2 and 3). The application runs as in the old environment, constructing a legacy-specific formatted output file (step 4). The wrapper then parses the specific output to generate generic (CSV or XML) formatted data files (step 5). The wrapper returns an exit code and the generic output file to the enterprise application for further processing (step 6). The key notion is that the enterprise application does not need to know any specific details of the legacy code. The communication layer follows a generic format (generic input arguments and generic output format), simplifying further processing of results.

FIGURE 6 Elements in plot data structure



C-MIST: Currents Measurements Interface for the Study of Tides - Netscape

http://cmist.co-ops.nos.noaa.gov/cmist/mysession/myplots.do

NOS/CO-OPS C-MIST Currents Measurement Interface for the Study of Tides

USER: pruessne
GROUP: admin

DATA SET: ak0303
SELECT: other data

My Plots

List is sortable by columns.

Plot Name	Phase	Station ID	Bin #	Plot Description	View Plot
preplot-0001.png	Pre-Analysis	AK0303	01	Percent good ping time series	view
preplot-0002.png	Pre-Analysis	AK0303	08	Correlation magnitude time series	view
preplot-0003.png	Pre-Analysis	AK0303	11	Velocity north/east time series	view
preplot-0004.png	Pre-Analysis	AK0303	all	Temperature and pressure time series *	view
preplot-0005.png	Pre-Analysis	AK0303	16	Echo intensity scatter plot time series	view

Attributes within data structure

File Edit View Go Bookmarks Tools Window Help
Home | Help | Search | Logout
Metadata Input | Data Ingestion | Data Requests | Processing | My Session | Public Reports
My Summary
My Plots
My Animations
My Analyses
My Reports

Modularity of Subsystems in Enterprise Applications

Enterprise applications involve many subsystems including databases, visualization modules, reporting tools, and analysis routines. Furthermore, they often are maintained over long periods of time with additional features or functionality added after the initial release. These additional features often extend beyond the original requirements specifications. The changing scope of enterprise applications over the product lifetime makes modular design a necessary feature of physical design. The advantages of modular design include:

- **Uniform APIs and exception handling.** As for legacy applications, the use of generic and uniform APIs to modules greatly simplifies maintenance and communication to and from each of the subsystems in the enterprise application. The uniform API and exception handling measures allow the subsystem-specific features to be encapsulated so the rest of the application does not need to be aware of the specifics of the particular subsystem.
- **Uniform data structures and formats.** Uniform data structures and data formats allow for simple addition of types to be added to the interface. For example, C-MIST has data structures representing generated plots. A history of plots generated can easily be displayed via the Web interface by just looping through the plot data structure. Figure 6 shows the elements within the plot data structure, which includes attributes such as plot name, plot phase, station/sensor ID, bin number (an identifier associated with depth), and plot description. If an analysis data structure is added then the analyses in the data structure can easily be displayed in the Web application by looping in a similar manner. No new code needs to be written to display contents of new objects that use the same data structure format.
- **Simplified long-term maintenance and enhancements.** Modularity of subsystems allows for simplified long-term maintenance of the enterprise application by encapsulating logic and potential errors to separable modules. In addition, it also allows for simple replacement of modules over time, which is particularly important if the legacy subsystems are to be replaced in the future. Often, the legacy components are replaced as resources become available and limitations of the legacy code become a bottleneck.

CONCLUSIONS

Although development on the C-MIST project is still in progress, the authors have shown how to migrate legacy applications to Web environments within an enterprise framework involving databases, Web front ends, and various other subsystems. Many issues are the same as for regular software quality assurance (particularly following a standard quality model); however, emphasis must be placed on key issues specific to legacy applications. These include verification of legacy results within the new environment, that is, does the software behave the same in the new environment as expected? Furthermore, generic API wrappers to the legacy components should be used. Generic APIs allow for encapsulation of legacy components and possible quality defects, consistent error and exception handling, and reusability within other applications. Modularity of subsystems is important in large-scale applications with many components (including legacy components) since subsystems can be replaced as needed with little impact on other subsystems. Note that it is common that legacy subsystems are eventually rewritten. Finally, the use of generic format output files can also simplify the replacement of subsystems or addition of new subsystems. Although the authors' case study involves a scientific computing application integrating legacy components into Web-application frameworks, the concepts are beneficial to all legacy integration projects.

Acknowledgment

The authors wish to thank the reviewers and the editor for their helpful comments, which greatly improved the original manuscript.

REFERENCES

Bennet, K. H. 1995. Legacy systems: Coping with success. *IEEE Software* 12, no. 1: 19-23.

Bisbal, J., D. Lawless, B. Wu, and J. Grimson. 1999. Legacy information systems: Issues and directions. *IEEE Software* 16, no. 5: 103-111.

Brodie, M., and M. Stonebraker. 1995. *Migrating legacy systems: Gateways, interfaces and the incremental approach.* San Francisco: Morgan Kaufman.

Burden, R. L., and J. D. Faires. 2005. *Numerical analysis*, 8th edition. London: Brookes-Cole.

Carnegie Mellon University, Software Engineering Institute. 1994. The *Capability Maturity Model: Guidelines for Improving the Software Process.* Reading, Mass.: Addison Wesley.

Laverty, J., C. Boldyreff, B. Ling, and C. Allison. 2004. Modeling the evolution of legacy systems to Web-based systems. J*ournal of Software Maintenance and Evolution: Research and Practic*e 16: 5-30.

Mittelmann, H., and A. Pruessner. 2006. A server for automated performance analysis and benchmarking of optimization software. *Optimization Methods and Software* 21: 105-120.

Paternostro, C., A. Pruessner, and R. Semkiw. 2005. Designing a quality oceanographic data processing environment. In *Proceedings of the MTS/IEEE Oceans 2005 Conference*, Washington D.C., September 18-23.

Pressman, R. S. 1997. *Software engineering: A practitioner's approach, 4th edition.* Boston: McGraw-Hill.

Schulmeyer, G. G., and J. I. McManus. 1999. *Handbook of software quality assurance*, 3rd Edition. Upper Saddle River, N. J.: Prentice Hall.

Schureman, P. 1940. Harmonic Analysis and Prediction of Tides. Special Publication no. 98, revised edition. Washington, D.C.: U.S. Department of Commerce, Coast and Geodetic Survey.

Software Engineering Institute. 2005a. *Welcome to the CMMI Website.* Pittsburgh: Carnegie Mellon University. Available online at http://www.sei.cmu.edu/cmmi/.

Software Engineering Institute. 2005b. *The Capability Maturity Model Integration (CMMI) Overview*, technical presentation. Pittsburgh: Carnegie Mellon University. Available online at http://www.sei.cmu.edu/cmmi/adoption/pdf/cmmi-overview05.pdf.

Zervas, C. 1999. Tidal current analysis procedures and associated computer programs. *Technical Report NOS CO-OPS 0021*, NOAA. Silver Spring, Md.: U.S. Department of Commerce.

Zetler, B. D. 1982. Computer applications to tides in the national ocean survey. Supplement to Manual of Harmonic Analysis and Prediction of Tides, special publication no. 98. Washington, D.C.: U.S. Department of Commerce, National Ocean Survey.

BIOGRAPHIES

Armin Pruessner is a scientific computing consultant currently developing large-scale oceanographic data processing systems for the National Oceanic and Atmospheric Administration (NOAA). He holds a master's degree in applied mathematics and scientific computing from the University of Maryland, College Park. He is an ASQ Certified Software Quality Engineer (CSQE) and has published numerous journal articles in areas ranging from numerical linear algebra and mathematical optimization to performance analysis and software quality. His research interests are in algorithms and scientific computing, with an emphasis on large-scale robust systems, useability, and quality. He can be reached by e-mail at Armin.Pruessner@noaa.gov .

Christopher L. Paternostro is an oceanographer with the National Oceanic and Atmospheric Administration and acting head of the National Current Observation Program. He holds a master's degree in oceanography from Texas A&M University. His research interests include coastal and estuarine circulation, computational hydrodynamic modeling, and numerical model development.